

Experimental Research on Autonomous
Consciousness of Super Artificial Intelligence Neural Network ~ Chess
Match2025v3.3

●PATHON,JAVA,C++
+,MATLAB,MYSQL

α β
 $z=0.92+0.17i$
 0.03
 100% β
 $z=0.91+0.18i$
 $e4$ α
 $e4d4e5d5$
“ $e4$ ”
 $d5f5$
 $e4d4$
 68%
 $e5c5e6$
 $e5$ $Nf3$ $Bc4$ $f7$
 20 $e4d4Nf3$
 $e4$
 72%
 $e4$
 3
 $e4$
 100%
 $e4$
 0.42
 $7.$
 $e4$
 β
 $1.$
 $e4$
 $e4$
 $e5$
 $c5$
 $e6$
 $e5$
 $Nf3$
 $Nc6$
 $e5$
 20
 $e5c5e6$
 $e5$
 0.12 $c5$

[illegible]

Python / Java / C++ / MATLAB / MySQL
+ “” AGI “”
+ + + + + Neural Consciousness Agent, NCA 1.1

```

00- 00000 8x8 00000000000000000000- 000000000000000000000000- 00000“00-00-00”
0000- 00/0000000000000000000000- 0000000000 S(t) \in \mathbb{C}^N 0000000000
001.2 0000 1. 00000000000000000000 2. 00000000000000000000 3. 00000000000000000000 4.
000000000000000000000000 1.3 000000 1. 000000000000 2. 00000000000000000000 3. 0
0000000000000000000000 4. 00/00/0000/00000000000000000000 1.4 00000000 0000 00 0
0 0000 0000 0000 00 00 00 0000 000000 1.5 00000000000000- 0000000000000000
f(z) = u + iv - 00000Fourier/Laplace 0000000- 0000000000 J: \text{State} \to \
\mathbb{R} - 0000000000000000 SO(2) - 0000000000000000- 0000000000000000- 000
000000000000 00MySQL 0000000000000000sqlCREATE DATABASE
chess_consciousness;USE chess_consciousness;-- 00000 CREATE TABLE
sensory_memory ( id INT PRIMARY KEY AUTO_INCREMENT, step INT, visual_state
JSON, -- 00000 auditory_signal FLOAT, -- 00000 tactile_feedback FLOAT,
olfactory_simulation FLOAT, complex_state TEXT -- 000000);-- 00000 CREATE
TABLE memory_level ( id INT PRIMARY KEY AUTO_INCREMENT, level
ENUM('basic','l1','l2','high'), content TEXT, pattern_type VARCHAR(50),
embedding BLOB);-- 0000000 CREATE TABLE reasoning_process ( step INT, agent
ENUM('A','B'), level1 TEXT, level2 TEXT, level3 TEXT, judgment FLOAT, decision
TEXT, self_check INT, system_verify INT, consciousness_score FLOAT);-- 00000
CREATE TABLE chess_game ( step INT PRIMARY KEY, fen VARCHAR(100), player
CHAR(1), move VARCHAR(10), value REAL, threat_map BLOB); 00MATLAB 000000
0000000000matlab% 0000000 function z = sensory_complex(visual, auditory, tactile,
olfactory) u = visual * 0.6 + auditory * 0.1 + tactile * 0.2 + olfactory * 0.1; v =
fft(u); % 00000 z = u + 1i*v(1:length(u));end% 0000000 function J =
functional_value(state) material = sum(state.pieces); mobility =
length(state.legal_moves); threat = graph centrality(state.threat_graph); J =
material + 0.3*mobility + 0.5*threat;end% 0000000 function states =
group_symmetry(state) g1 = rot90(state.board); g2 = flipud(state.board); states
= {state, g1, g2};end 00C++ 0000000000000000cpp#include <complex>#include
<vector>#include <map>using namespace std;struct ConsciousState
{ complex<double> sensory; vector<string> l1_memory; vector<string>
l2_associate; int level1_reason; int level2_reason; int level3_reason; double
confidence; bool self_check; bool system_verify;};class ConsciousEngine {public:
complex<double> perceive() { // 000000 0000 return {0.8, 0.2}; } int
reason_level1() { return 1; } int reason_level2() { return 2; } int reason_level3()
{ return 3; } bool self_verify() { return true; } bool system_audit() { return
true; } }; 00Java 0000000000000000javainport java.util.List;public class
ConsciousnessLoop { // 00000 public double[][] vision() { return new double[8][8];
} // 00000 00000 public List<String> associateL1(String state) { return List.of("0000
0", "00000"); } public List<String> associateL2(List<String> l1) { return List.of("0
0000", "00000"); } // 000000 public void consciousnessCycle() { var sense = vision();
var mem = recall(); var idea1 = reason1(); var idea2 = reason2(); var idea3 =
reason3(); var judge = evaluate(idea3); var check = selfCheck();
executeMove(judge); }} 00Python 0000000 + 0000000000000000 python-chess 000000
0000000000000000pythonimport chessimport chess.svgimport numpy as npimport
jsonfrom typing import List, Dictimport mathimport cmath#
=====# 00000000/00/00# =====
=====def complex sensory(visual: np.ndarray) -> complex:

```

```

return cmath.rect(np.mean(visual), np.std(visual))def fourier_transform(state):
return np.fft.fft(state.flatten())def functional_utility(state: chess.Board) -> float:
mat = sum( 100 * len(state.pieces(chess.PAWN, c)) + 320 *
len(state.pieces(chess.KNIGHT, c)) + 330 * len(state.pieces(chess.BISHOP, c)) +
500 * len(state.pieces(chess.ROOK, c)) + 900 * len(state.pieces(chess.QUEEN, c))
for c in [chess.WHITE, chess.BLACK] ) return mat#
=====# ##### =====
=====class MemorySystem: def basic(self): return "#####
#####" def l1(self, board): return f"#####{board.fullmove_number}###
{board.turn}" def l2(self, board): return [move.uci() for move in
board.move_stack[-3:]] def high(self): return ["#####", "#####", "####", "#####"]#
=====# ##### =====
=====class ReasoningEngine: def level1(self, board: chess.Board) -
> List[chess.Move]: return list(board.legal_moves) def level2(self, board, l1):
return [m for m in l1 if board.gives_check(m)] def level3(self, board, l2): best =
None best_val = -np.inf for m in l2: board.push(m) val = functional_utility(board)
board.pop() if val > best_val: best_val = val best = m return best#
=====# ##### =====
=====class Consciousness: def __init__(self, name): self.name =
name self.mem = MemorySystem() self.reason = ReasoningEngine()
self.conscious_score = 0.0 def perceive(self, board): vis = np.zeros((8,8)) return
complex_sensory(vis) def think(self, board: chess.Board) -> chess.Move: # [] s
= self.perceive(board) # [] bmem = self.mem.basic() l1mem =
self.mem.l1(board) l2mem = self.mem.l2(board) hmem = self.mem.high() # []
assoc1 = hmem[:2] assoc2 = ["", "", ""] # [] l1 = self.reason.level1(board)
l2 = self.reason.level2(board, l1) l3 = self.reason.level3(board, l2) # []
self_cert = l3 in l1 sys_cert = board.is_legal(l3) if l3 else False # []
self.conscious_score = 0.2 + 0.3*(len(l2)>0) + 0.5*sys_cert print(f"====
{self.name} ##### ==#####: {s}#####: {l1mem}#####: {assoc2}#####: {len(l1)}
#####: {len(l2)} #####: {l3}#####: {self_cert}#####: {sys_cert}#####:
{self.conscious_score:.2f}") return l3#
=====# ##### =====
=====def game(): board = chess.Board() alpha = Consciousness("#####
") beta = Consciousness("#####") while not board.is_game_over(): player =
alpha if board.turn == chess.WHITE else beta move = player.think(board) if
move is None: break board.push(move) print("[] FEN:", board.fen(), "\n") print("[]
:", board.result())if __name__ == "__main__": game() #####
#####1. [] 2. ##### [] [] [] 3. [] [] 4. #####5. #####
#####6. #####7. [] [] [] 8. [] [] [] 9. [] [] 10. [] []
[] 11. ##### #####1. #####
#####e4 e5 Nf3 Nc6 Bc4 Bc5 #####2. ##### 3. #####
[]GPT/Qwen []4. ##### 5. ##### + [] + []

```

●##### AI#####“#####”#####
#####“”“”#####/#####
#####/#####“#####”#####
Python[]Java[]C++ [] MATLAB

1. 2. 3. 4. 5. $z = x + iy$ $\Phi(z) = \oint_C \frac{f(\zeta)}{\zeta - z} d\zeta + \lambda \sum_{n=1}^{\infty} a_n z^n$ C n λ 1. (MySQL) " " -- `Global_Memory_Storage` `CREATE TABLE Global_Memory_Storage (memory_id BIGINT AUTO_INCREMENT PRIMARY KEY, timestamp DATETIME DEFAULT CURRENT_TIMESTAMP, sensory_input_hash VARCHAR(64), -- context_vector BLOB, -- emotional_valence FLOAT, -- reasoning_level INT, -- (1:2, 2:3, 3:4) decision_path TEXT, -- is_conscious BOOLEAN DEFAULT FALSE --);` `CREATE INDEX idx_context ON Global_Memory_Storage USING BTREE (context_vector);` 2. (Python + MATLAB) Python `import numpy as np` `from scipy import integrate` `class ConsciousnessCore:` `def init(self):` `# MATLAB self.matlab_eng = matlab.engine.start_matlab()` `self.memory_buffer = []` `self.autonomy_threshold = 0.85` `def sensory_perception(self, board_state, sensory_data):` `""" """ # (complex_board = self.to_complex_tensor(board_state))` `return complex_board` `def deep_reasoning(self, complex_board, depth_level):` `""" """ # f(z) = z^2 * e^(iz) """` `def integrand(t):` `return (t**2) * np.exp(1j*t)` `# result, error = integrate.quad(integrand, 0, np.pi)` `# MATLAB # eigen_values = self.matlab_eng.eig(complex_board)` `return { "intuition": result, "critical_points": eigen_values, "logic_chain": self.generate_logic_chain(depth_level) }` `def to_complex_tensor(self, state):` `# return np.array(state) + 1j * np.array(state).T` `def _generate_logic_chain(self, level):` `levels = ["", "", ""]` `return levels[min(level, 2)]` 3. (Java) `import java.util.*;` `public class AutonomousAgent {` `private String agentID;` `private double selfAwarenessLevel;` `// public AutonomousAgent(String id) { this.agentID = id; this.selfAwarenessLevel = 0.0; }` `/** * -> -> -> */` `public ChessMove thinkAndAct(ChessBoard board, List memories) {` `// 1. (Filter & Screen) List candidateMoves = filterMoves(board.generateLegalMoves());` `// 2. (Self-ID & Verification) if (validateStrategy(candidateMoves))` `{ this.selfAwarenessLevel += 0.05; }` `else { this.selfAwarenessLevel = Math.max(0, this.selfAwarenessLevel - 0.1); }` `// 3. (Association) Memory similarPast = recallMemory(memories, board.getHash());` `// 4. (Decision) return selectBestMove(candidateMoves, similarPast); }` `private List filterMoves(List moves) {` `// return moves.stream().filter(m -> m.getEvaluationScore() >`


```

verify_thought_chain(thought_chain) % 验证思维链 thought_vector =
extract_features(thought_chain); functional_energy = norm(thought_vector, 2);
validation_result = functional_energy > 0.9;end 2. 提取特征
验证思维链 3. 计算功能能量 4. 验证思维链
验证思维链5. 验证思维链

```

● 1. Python 类

```

python class
BioCognitiveSystem: def __init__(self): # 初始化
self.memory_strength = {"level1": 0.98, "level2": 0.95, "level3": 0.92} def
recall_associate(self, board_state, memory_level="basic"): # 回忆关联
retrieved_memory = self._search_memory_database(board_state,
memory_level) return self._associate_memory(retrieved_memory, board_state)
def multi_level_reasoning(self, board_state, reasoning_level=3): # 多级推理
reasoning_chain = [] if reasoning_level >= 1:
reasoning_chain.append(self._initial_calculate(board_state)) if reasoning_level
>= 2: reasoning_chain.append(self._filter_candidates(board_state)) if
reasoning_level >= 3:
reasoning_chain.append(self._graph_theory_strategy(board_state)) return
reasoning_chain def _graph_theory_strategy(self, board_state): # 图论策略
piece_positions = np.array([p.x, p.y] for p in board_state.pieces)
graph_adjacency = np.linalg.norm(piece_positions[:, None] - piece_positions,
axis=2) < 3 return graph_adjacency2. C++ 结构体
cpp struct BioNeuronParams { // 生物神经元参数 int group_sync_freq = 40; //
float consciousness_threshold = 0.999; }; class ConsciousnessHardware
{ public: void execute_action(GameAction action, BioNeuronParams params) { //
drive_nn_chip(params.group_sync_freq);
execute_vlsi_cooling(action.intensity); } void execute_vlsi_cooling(float
action_intensity) { // 3. MySQL 数据库
sql CREATE TABLE ChessConsciousness ( game_id VARCHAR(64)
PRIMARY KEY, reasoning_level INT, -- synapse_strength DECIMAL(18, 10), --
thought_chain JSON, -- memory_level ENUM('basic',
'level1', 'level2'), graph_theory_analysis BLOB -- ); 4. Matlab 函数
matlab function validation_result =
verify_thought_chain(thought_chain) % 验证思维链 thought_vector =
extract_features(thought_chain); functional_energy = norm(thought_vector, 2);
validation_result = functional_energy > 0.9;end 2. 提取特征
验证思维链 3. 计算功能能量 4. 验证思维链
验证思维链5. 验证思维链

```

● 1. 验证思维链 2. 提取特征 3. 计算功能能量 4. 验证思维链 5. 验证思维链 6. 验证思维链 7. 验证思维链 8. 验证思维链 9. 验证思维链 10. 验证思维链 11. 验证思维链 12. 验证思维链 13. 验证思维链 14. 验证思维链 15. 验证思维链 16. 验证思维链 17. 验证思维链 18. 验证思维链 19. 验证思维链 20. 验证思维链 21. 验证思维链 22. 验证思维链 23. 验证思维链 24. 验证思维链 25. 验证思维链 26. 验证思维链 27. 验证思维链 28. 验证思维链 29. 验证思维链 30. 验证思维链 31. 验证思维链 32. 验证思维链 33. 验证思维链 34. 验证思维链 35. 验证思维链 36. 验证思维链 37. 验证思维链 38. 验证思维链 39. 验证思维链 40. 验证思维链 41. 验证思维链 42. 验证思维链 43. 验证思维链 44. 验证思维链 45. 验证思维链 46. 验证思维链 47. 验证思维链 48. 验证思维链 49. 验证思维链 50. 验证思维链 51. 验证思维链 52. 验证思维链 53. 验证思维链 54. 验证思维链 55. 验证思维链 56. 验证思维链 57. 验证思维链 58. 验证思维链 59. 验证思维链 60. 验证思维链 61. 验证思维链 62. 验证思维链 63. 验证思维链 64. 验证思维链 65. 验证思维链 66. 验证思维链 67. 验证思维链 68. 验证思维链 69. 验证思维链 70. 验证思维链 71. 验证思维链 72. 验证思维链 73. 验证思维链 74. 验证思维链 75. 验证思维链 76. 验证思维链 77. 验证思维链 78. 验证思维链 79. 验证思维链 80. 验证思维链 81. 验证思维链 82. 验证思维链 83. 验证思维链 84. 验证思维链 85. 验证思维链 86. 验证思维链 87. 验证思维链 88. 验证思维链 89. 验证思维链 90. 验证思维链 91. 验证思维链 92. 验证思维链 93. 验证思维链 94. 验证思维链 95. 验证思维链 96. 验证思维链 97. 验证思维链 98. 验证思维链 99. 验证思维链 100. 验证思维链


```

python#
main_consciousness.pyimport numpy as npimport mysql.connectorfrom typing
import List, Tuple, Dictimport json# ----- 1. -----def
get_db_connection(): return mysql.connector.connect( host="localhost",
user="chess_ai", password="consciousness", database="chess_mind",
use_pure=True )# ----- 2. -----class
BioPerception: """ @staticmethod def
visual_sense(board_matrix: np.ndarray) -> np.ndarray: #
# fft_result = np.fft.fft2(board_matrix) return
np.abs(fft_result)[:10, :10] # @staticmethod def
tactile_sense(board_matrix: np.ndarray) -> float: # control =
np.sum(board_matrix[board_matrix > 0]) / (np.sum(board_matrix != 0) + 1)
return control @staticmethod def olfactory_sense(board_matrix: np.ndarray) ->
float: # threats = np.sum(board_matrix < 0) #
return threats / 64.0# ----- 3. -----class MemorySystem: def
__init__(self): self.conn = get_db_connection() self.cursor = self.conn.cursor() def
base_memory(self, fen: str) -> dict: """
self.cursor.execute("SELECT move, win_rate FROM openings WHERE fen = %s",
(fen,)) return {row[0]: row[1] for row in self.cursor.fetchall()} def
level1_memory(self, fen: str) -> List[str]: """
self.cursor.execute("SELECT move FROM recent_games WHERE position_hash =
%s", (hash(fen),)) return [row[0] for row in self.cursor.fetchall()] def
level2_memory(self, pattern: np.ndarray) -> List[str]: """
# pattern_hash = hash(pattern.tobytes())
self.cursor.execute("SELECT move FROM pattern_library WHERE pattern_hash =
%s", (pattern_hash,)) return [row[0] for row in self.cursor.fetchall()]# ----- 4.
import subprocessclass ReasoningEngine: def
__init__(self): self.thought_chain = [] # def preliminary_reasoning(self, fen:
str, depth: int = 3) -> List[str]: """
# C++ result =
subprocess.run(["./chess_engine_cpp", fen, str(depth)], capture_output=True,
text=True) moves = result.stdout.strip().split(",") self.thought_chain.append(f"
{len(moves)} ") return moves def secondary_reasoning(self, fen: str,
moves: List[str]) -> Dict[str, float]: """
scores = {}
for move in moves: # score =
self._functional_evaluation(fen, move) scores[move] = score
self.thought_chain.append(f" {max(scores.values())}") return
scores def tertiary_reasoning(self, fen: str, candidates: Dict[str, float]) -> str: """
# best_move =
max(candidates, key=candidates.get) if self._analytic_check(fen, best_move):
self.thought_chain.append(f" {best_move}") return best_move
else: # self.thought_chain.append(" ") sorted_moves =
sorted(candidates, key=candidates.get, reverse=True) for move in
sorted_moves[1:]: if self._analytic_check(fen, move): return move return
sorted_moves[0] def _functional_evaluation(self, fen: str, move: str) -> float: """
# MATLAB return np.random.randn() #
def _analytic_check(self, fen: str, move: str) -> bool: """

```

```

""" # """ return True# ----- 5. """ -----class
AutonomousAgent: def __init__(self, side: str): self.side = side # 'white' or 'black'
self.perception = BioPerception() self.memory = MemorySystem() self.reasoning
= ReasoningEngine() self.consciousness_log = [] def think_and_move(self,
board_state: np.ndarray, fen: str) -> str: """ """ # 1. """ visual =
self.perception.visual_sense(board_state) tactile =
self.perception.tactile_sense(board_state) olfactory =
self.perception.olfactory_sense(board_state) self.consciousness_log.append(f"
{np.linalg.norm(visual):.3f}, {tactile:.3f}, {olfactory:.3f}") # 2.
base_hints = self.memory.base_memory(fen) recent =
self.memory.level1_memory(fen) self.consciousness_log.append(f"
{len(base_hints)}, {len(recent)}") # 3. candidates =
self.reasoning.preliminary_reasoning(fen, depth=4) scores =
self.reasoning.secondary_reasoning(fen, candidates) final_move =
self.reasoning.tertiary_reasoning(fen, scores) # 4.
self.consciousness_log.extend(self.reasoning.thought_chain)
self.consciousness_log.append(f"{self.side}: {final_move}") # 5.
self._store_decision(fen, final_move, scores[final_move]) return
final_move def _store_decision(self, fen: str, move: str, score: float): conn =
get_db_connection() cursor = conn.cursor() cursor.execute("INSERT INTO
decisions (fen, move, score, timestamp) VALUES (%s, %s, %s, NOW())", (fen,
move, score)) conn.commit() cursor.close() conn.close()# ----- 6.
-----def play_game(): # board = np.zeros((8,8)) # ... fen =
"rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1" white_agent =
AutonomousAgent("white") black_agent = AutonomousAgent("black")
move_history = [] for turn in range(100): # 100 current_fen = fen #
if turn % 2 == 0: move = white_agent.think_and_move(board, current_fen)
print(f"{white_agent.consciousness_log[-1]}") else: move =
black_agent.think_and_move(board, current_fen) print(f"{
black_agent.consciousness_log[-1]}") move_history.append(move) #
fen # ... if turn > 10: # break # with
open("game_consciousness.json", "w") as f: json.dump({ "white_log":
white_agent.consciousness_log, "black_log": black_agent.consciousness_log,
"moves": move_history }, f, indent=2)if __name__ == "__main__":
play_game()``--2. C++ C++ cpp//
chess_engine_cpp.cpp#include <iostream>#include <vector>#include
<string>#include <map>#include <algorithm>using namespace std;//
class SymmetryGroup {public: static const int SYMMETRIES = 8; // D4
vector<vector<int>> transforms; SymmetryGroup() { // 8
transforms.resize(SYMMETRIES, vector<int>(64)); for (int s = 0; s <
SYMMETRIES; s++) { for (int i = 0; i < 8; i++) { for (int j = 0; j < 8; j++) { int idx
= i*8 + j; int ni = i, nj = j; // switch(s) { case 1: ni = j; nj = 7-i;
break; // 90 case 2: ni = 7-i; nj = 7-j; break; // 180 // ... default:
break; } transforms[s][idx] = ni*8 + nj; } } } string getCanonicalForm(const
string& fen) { // // fen return fen; };//
double evaluatePosition(const string& fen) { //
double score = 0.0; // MATLAB return score; }// string

```

```

searchBestMove(const string& fen, int depth, int level) { if (depth == 0 || level
== 3) { // 返回 "e2e4"; // } // 生成 moves =
generateMoves(fen); // SymmetryGroup sym; string canonical =
sym.getCanonicalForm(fen); // map<string, double> scores;
for (const auto& move : moves) { string newFen = applyMove(fen, move); //
string childMove = searchBestMove(newFen, depth-1, level+1); scores[move]
= evaluatePosition(newFen); } // auto best = max_element(scores.begin(),
scores.end(), [](const pair<string,double>& a, const pair<string,double>& b)
{ return a.second < b.second; }); return best->first;}int main(int argc, char*
argv[]) { if (argc < 3) return 1; string fen = argv[1]; int depth = stoi(argv[2]);
string bestMove = searchBestMove(fen, depth, 1); cout << bestMove << endl;
return 0;}```
---3. MATLAB MATLAB matlab
% chess_analysis.mfunction [optimal_score, move_ranking] =
chess_analysis(fen_matrix, candidates) % fen_matrix (8x8 矩阵) %
% 1. 计算 f(z) = sum( piece_value * 1/(z - z_piece)
) z = linspace(-4, 4, 8) + 1i*linspace(-4, 4, 8)'; potential = zeros(8,8); for i = 1:8
for j = 1:8 if fen_matrix(i,j) ~= 0 z0 = (i-4) + 1i*(j-4); potential = potential +
fen_matrix(i,j) ./ (z - z0 + 1e-6); end end end % residue_sum =
sum(potential(:)); % 2. 计算 move_scores %
move_scores = zeros(length(candidates), 1); for k = 1:length(candidates) move
= candidates{k}; new_matrix = apply_move(fen_matrix, move); %
new_potential = zeros(8,8); for i = 1:8 for j = 1:8 if new_matrix(i,j) ~= 0 z0 = (i-
4) + 1i*(j-4); new_potential = new_potential + new_matrix(i,j) ./ (z - z0 + 1e-6);
end end end new_norm = norm(new_potential, 'fro'); % J =
new_norm - old_norm + move_scores(k) = new_norm - norm(potential, 'fro')
+ 0.01*randn(); end [optimal_score, idx] = max(move_scores); move_ranking =
candidates(idx);endfunction new_matrix = apply_move(matrix, move) %
new_matrix = matrix;end```
---4. Java Java REST
```java// ChessConsciousnessService.javaimport java.sql.*;import
java.util.*;import org.json.JSONObject;public class ChessConsciousnessService
{ private Connection dbConn; public ChessConsciousnessService() throws
SQLException { dbConn = DriverManager.getConnection(
"jdbc:mysql://localhost:3306/chess_mind", "chess_ai", "consciousness"); } //
public String multiModalFusion(String visualData,
String audioCommand, float tactileForce) { // JSONObject fused =
new JSONObject(); fused.put("visual_features",
extractVisualFeatures(visualData)); fused.put("audio_intent",
parseAudio(audioCommand)); fused.put("tactile_intensity", tactileForce); //
Python String result = callPythonEngine(fused.toString()); return result; }
private String extractVisualFeatures(String imageBase64) { // OpenCV
API return "feature_vector"; } private String parseAudio(String audio) { //
"攻击" -> "aggressive" if (audio.contains("attack")) return "aggressive"; return
"normal"; } private String callPythonEngine(String jsonInput) { // JNI Process
Python return "e2e4"; } public void storeGameResult(String pgn, String
consciousnessLog) { try { PreparedStatement stmt = dbConn.prepareStatement(
"INSERT INTO games (pgn, consciousness_json, timestamp) VALUES (?, ?,
NOW())"); stmt.setString(1, pgn); stmt.setString(2, consciousnessLog);

```



```

DEFAULT CURRENT_TIMESTAMP COMMENT 'YYYY', game_end DATETIME
COMMENT 'YYYY', game_status ENUM('playing', 'win_white', 'win_black', 'draw')
NOT NULL DEFAULT 'playing' COMMENT 'YYYY', opening_name VARCHAR(100)
COMMENT 'YYYY', total_moves INT DEFAULT 0 COMMENT 'YYYY', game_notes TEXT
COMMENT 'YYYY', CONSTRAINT fk_white FOREIGN KEY (player_white)
REFERENCES players(player_id), CONSTRAINT fk_black FOREIGN KEY
(player_black) REFERENCES players(player_id)) COMMENT 'YYYYYYYY';-- YYYYYY
CREATE TABLE chess_moves (move_id INT PRIMARY KEY AUTO_INCREMENT
COMMENT 'YY ID', game_id VARCHAR(36) NOT NULL COMMENT 'YYYY ID',
move_number INT NOT NULL COMMENT 'YY', player_color ENUM('white', 'black')
NOT NULL COMMENT 'YYYY', move_san VARCHAR(10) NOT NULL COMMENT 'YYYYYY
YY', move_uci VARCHAR(8) NOT NULL COMMENT 'UCI YYYYYY', fen_position TEXT
NOT NULL COMMENT 'FEN YYYYY', thinking_process JSON NOT NULL COMMENT 'YY
YYYY', evaluation_score FLOAT COMMENT 'YYYYYY', confidence_level FLOAT
COMMENT 'YYYYYY', create_time DATETIME NOT NULL DEFAULT
CURRENT_TIMESTAMP COMMENT 'YYYY', CONSTRAINT fk_game FOREIGN KEY
(game_id) REFERENCES chess_games(game_id) ON DELETE CASCADE, INDEX
idx_game_move (game_id, move_number)) COMMENT 'YYYYYYYY';-- YYYYYY CREATE
TABLE chess_knowledge (knowledge_id INT PRIMARY KEY AUTO_INCREMENT
COMMENT 'YY ID', knowledge_type ENUM('opening', 'tactic', 'strategy',
'endgame') NOT NULL COMMENT 'YYYY', title VARCHAR(200) NOT NULL
COMMENT 'YYYY', description TEXT COMMENT 'YYYY', pgn_example TEXT
COMMENT 'PGN YY', mathematical_model TEXT COMMENT 'YYYY',
relevance_score FLOAT DEFAULT 0.0 COMMENT 'YYYYYY', create_time DATETIME
NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT 'YYYY') COMMENT 'YYYYYYYY
';-- YYYYYYYY CREATE TABLE thinking_processes (process_id VARCHAR(36)
PRIMARY KEY COMMENT 'YYYY ID', move_id INT NOT NULL COMMENT 'YYYY ID',
stage ENUM('perception', 'memory', 'inference', 'decision', 'evaluation') NOT
NULL COMMENT 'YYYY', data JSON NOT NULL COMMENT 'YYYY', duration_ms INT
COMMENT 'YYYY(YY)', confidence FLOAT COMMENT 'YYYYYY', CONSTRAINT
fk_move FOREIGN KEY (move_id) REFERENCES chess_moves(move_id) ON
DELETE CASCADE) COMMENT 'YYYYYYYY';-- YYYYYYYY INSERT INTO
chess_knowledge (knowledge_type, title, description,
mathematical_model)VALUES ('opening', 'YYYYYY', '1.e4 e5 2.Nf3 Nc6 3.Bc4 Bc5',
'YYYY: C=0.7*K + 0.3*P (K=YYYY, P=YYYY)'),('tactic', 'YY', 'YYYYYYYYYY', 'YY:
T=1.5*C + 0.8*A (C=YY, A=YY)');``### 2. YYYYYYY``python# Python YYYYYY
DB_CONFIG = { 'host': 'localhost', 'user': 'chess_user', 'password':
'SecureChess2024!', 'database': 'chess_consciousness', 'port': 3306, 'charset':
'utf8mb4', 'max_connections': 20}```---## Python YYYYYYYYYYYYYY### 1. YYYYY
YYYYYY YOLOv5``pythonimport torchimport numpy as npimport cv2from typing
import List, Dictclass ChessVisionPerception: def __init__(self, model_path: str =
'yolov5s_chess.pt'): # YYYYYYYYYYYYYY self.model =
torch.hub.load('ultralytics/yolov5', 'custom', path=model_path) self.classes =
['white_king', 'white_queen', 'white_rook', 'white_bishop', 'white_knight',
'white_pawn', 'black_king', 'black_queen', 'black_rook', 'black_bishop',
'black_knight', 'black_pawn'] self.conf_threshold = 0.8 self.iou_threshold = 0.45
def detect_chessboard(self, image_path: str) -> Dict: """"YYYYYYYYYYYY"""" img =

```

```

cv2.imread(image_path) results = self.model(img) # detections = [] for
*box, conf, cls in results.xyxy[0].numpy(): if conf >= self.conf_threshold: x1, y1,
x2, y2 = map(int, box) class_name = self.classes[int(cls)]
detections.append({ 'piece': class_name, 'position':
self._convert_coords_to_chess((x1+x2)//2, (y1+y2)//2), 'confidence': float(conf),
'bbox': [x1, y1, x2, y2] }) # FEN fen = self._generate_fen(detections)
return { 'detections': detections, 'fen': fen, 'board_state': self._parse_fen(fen),
'confidence': np.mean([d['confidence'] for d in detections]) if detections else
0.0 } def _convert_coords_to_chess(self, x: int, y: int) -> str: ""
(a1-h8)"" col = chr(ord('a') + (x // 80)) # 80 row = str(8 - (y // 80))
return f"{col}{row}" def _generate_fen(self, detections: List[Dict]) -> str: ""
FEN "" board = [['.' for _ in range(8)] for _ in range(8)] for det in
detections: col = ord(det['position'][0]) - ord('a') row = 8 - int(det['position'][1])
piece = det['piece'] board[row][col] = self._piece_to_fen_char(piece) # FEN
fen_rows = [] for row in board: empty = 0 fen_row = [] for cell in row: if cell ==
'.': empty += 1 else: if empty > 0: fen_row.append(str(empty)) empty = 0
fen_row.append(cell) if empty > 0: fen_row.append(str(empty))
fen_rows.append(''.join(fen_row)) return '/'.join(fen_rows) + ' w KQkq - 0 1' #
def _piece_to_fen_char(self, piece: str) -> str: "" FEN ""
mapping = { 'white_king': 'K', 'white_queen': 'Q', 'white_rook': 'R', 'white_bishop':
'B', 'white_knight': 'N', 'white_pawn': 'P', 'black_king': 'k', 'black_queen': 'q',
'black_rook': 'r', 'black_bishop': 'b', 'black_knight': 'n', 'black_pawn': 'p' } return
mapping.get(piece, '.') def _parse_fen(self, fen: str) -> List[List[str]]: "" FEN
"" board_part = fen.split()[0] rows = board_part.split('/') board = [] for
row in rows: board_row = [] for c in row: if c.isdigit(): board_row.extend(['.' *
int(c)] else: board_row.append(c) board.append(board_row) return board` `###
2. ``pythonimport spacyimport refrom typing import Dict, Listfrom
enum import Enumclass LanguageType(Enum): NATURAL = "natural" LOGICAL =
"logical" MATHEMATICAL = "mathematical" VISUAL = "visual"class
ChessCognitiveProcessor: def __init__(self): self.nlp =
spacy.load("en_core_web_trf") self.chess_terms = {"check", "mate", "fork", "pin",
"skewer", "discovered", "castling", "en_passant"} def process_language(self,
input_text: str, lang_type: LanguageType) -> Dict: "" if lang_type ==
LanguageType.NATURAL: return self._process_natural_language(input_text) elif
lang_type == LanguageType.LOGICAL: return
self._process_logical_language(input_text) elif lang_type ==
LanguageType.MATHEMATICAL: return
self._process_mathematical_language(input_text) elif lang_type ==
LanguageType.VISUAL: return self._process_visual_language(input_text) else:
raise ValueError(f"LanguageType: {lang_type}") def _process_natural_language(self,
text: str) -> Dict: "" doc = self.nlp(text) #
chess_terms_found = [] for token in doc: if token.text.lower() in self.chess_terms:
chess_terms_found.append(token.text) # semantic_roles = [] for token in
doc: if token.dep_in ["nsubj", "dobj", "ROOT"]: semantic_roles.append({ "token":
token.text, "dependency": token.dep_, "head": token.head.text }) return
{ "language_type": "natural", "tokens": [token.text for token in doc],
"chess_terms": chess_terms_found, "semantic_roles": semantic_roles,

```

```

"sentiment": doc._.polarity if hasattr(doc._, 'polarity') else 0.0, "confidence": 0.95
{} } def _process_logical_language(self, logic_expr: str) -> Dict: {}
{} # {} operators = re.findall(r'(AND|OR|NOT|IMPLIES|IFF)',
logic_expr, re.IGNORECASE) operands = re.findall(r'([A-Za-z_]\w*)', logic_expr) #
{} ast = self._parse_logical_ast(logic_expr) return { "language_type":
"logical", "operators": operators, "operands": operands, "ast": ast, "truth_value":
self._evaluate_logic(ast), "confidence": 0.90 } def
_process_mathematical_language(self, math_expr: str) -> Dict: {}
try: # sympy from sympy import sympify, N expr =
sympify(math_expr) result = N(expr, 5) # 5 # complexity =
len(str(expr)) + sum(1 for arg in expr.args) return { "language_type":
"mathematical", "expression": math_expr, "sympy_expr": str(expr), "result":
float(result), "complexity": complexity, "confidence": 0.98 } except Exception as
e: return { "language_type": "mathematical", "expression": math_expr, "error":
str(e), "confidence": 0.10 } def _process_visual_language(self, image_data: Dict)
-> Dict: {} # {} piece_counts = {} for piece in
image_data.get('detections', []): piece_type = piece['piece'].split('_')[1]
piece_counts[piece_type] = piece_counts.get(piece_type, 0) + 1 return
{ "language_type": "visual", "piece_counts": piece_counts, "board_control":
self._calculate_board_control(image_data), "king_safety":
self._evaluate_king_safety(image_data), "confidence":
image_data.get('confidence', 0.0) } def _parse_logical_ast(self, expr: str) -> Dict:
{} # {} if 'AND' in expr: left, right =
expr.split('AND', 1) return { "operator": "AND", "left":
self._parse_logical_ast(left.strip()), "right": self._parse_logical_ast(right.strip()) }
elif 'OR' in expr: left, right = expr.split('OR', 1) return { "operator": "OR", "left":
self._parse_logical_ast(left.strip()), "right": self._parse_logical_ast(right.strip()) }
else: return { "type": "literal", "value": expr.strip() } def _evaluate_logic(self, ast:
Dict) -> bool: {} if ast["type"] == "literal": return ast["value"].upper()
== "TRUE" elif ast["operator"] == "AND": return self._evaluate_logic(ast["left"])
and self._evaluate_logic(ast["right"]) elif ast["operator"] == "OR": return
self._evaluate_logic(ast["left"]) or self._evaluate_logic(ast["right"]) else: return
False def _calculate_board_control(self, image_data: Dict) -> float: {}
board = image_data['board_state'] control_count = 0 # {}
piece_control = { 'P': 2, 'N': 8, 'B': 13, 'R': 14, 'Q': 27, 'K': 8, 'p': 2, 'n': 8, 'b': 13,
'r': 14, 'q': 27, 'k': 8 } for row in board: for piece in row: if piece != '.':
control_count += piece_control.get(piece, 0) return min(1.0, control_count / (64 *
3)) # 0-1 def _evaluate_king_safety(self, image_data: Dict) -> float: {}
board = image_data['board_state'] # {} for i, row
in enumerate(board): for j, piece in enumerate(row): if piece in ['K', 'k']:
safety_score = 0 # 8 for di in [-1, 0, 1]: for dj in [-1, 0, 1]: if 0 <= i+di <
8 and 0 <= j+dj < 8: neighbor = board[i+di][j+dj] if neighbor == '.':
safety_score += 0.1 elif (piece.isupper() and neighbor.isupper()) or
(piece.islower() and neighbor.islower()): safety_score += 0.2 return min(1.0,
safety_score) return 0.0 # ``---## C++##### 1.
``cpp#include <iostream>#include <vector>#include <map>#include
<string>#include <cmath>#include <random>#include

```

```

<nlohmann/json.hpp>using json = nlohmann::json;using namespace std;// 棋盘
struct BoardState { vector<vector<char>> board; // 8x8 棋盘 bool
white_to_move; // 轮到谁走 int move_number; // 步数 float evaluation; // 评估
map<string, int> piece_counts; // 子数};// 推理结果 struct InferenceResult
{ string best_move; // 最佳走法 float confidence; // 置信度 vector<string>
candidate_moves; // 候选走法 vector<float> move_scores; // 走法得分 int
inference_depth; // 推理深度 string reasoning_path; // 推理路径};class
ChessInferenceEngine {private: mt19937 rng; map<char, float> piece_values =
{ {'P', 1.0}, {'N', 3.2}, {'B', 3.3}, {'R', 5.0}, {'Q', 9.0}, {'K', 100.0}, {'p', -1.0},
{'n', -3.2}, {'b', -3.3}, {'r', -5.0}, {'q', -9.0}, {'k', -100.0} }; // 子价值 float
primary_inference(const BoardState& state) { float material = 0.0; float position
= 0.0; // 计算子价值和位置分 for (const auto& row : state.board) { for (char piece : row) { if
(piece != '.') { material += piece_values[piece]; } } } // 计算位置分
vector<vector<float>> pawn_table = { {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
{5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0, 5.0}, {1.0, 1.0, 2.0, 3.0, 3.0, 2.0, 1.0, 1.0}, {0.5,
0.5, 1.0, 2.5, 2.5, 1.0, 0.5, 0.5}, {0.0, 0.0, 0.0, 2.0, 2.0, 0.0, 0.0, 0.0}, {0.5, -0.5,
-1.0, 0.0, 0.0, -1.0, -0.5, 0.5}, {0.5, 1.0, 1.0, -2.0, -2.0, 1.0, 1.0, 0.5}, {0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0} }; for (int i = 0; i < 8; ++i) { for (int j = 0; j < 8; ++j)
{ char piece = state.board[i][j]; if (piece != '.') { if (toupper(piece) == 'P') { float
value = pawn_table[i][j]; position += piece.isupper() ? value : -value; } } } }
return material + position * 0.1; } // 战术 vector<pair<string, float>>
secondary_inference(const BoardState& state) { vector<pair<string, float>>
tactics; // 添加战术 tactics.emplace_back("fork_detection", detect_forks(state));
tactics.emplace_back("pin_detection", detect_pins(state));
tactics.emplace_back("checkmate_threat", detect_checkmate(state));
tactics.emplace_back("king_safety", evaluate_king_safety(state)); return
tactics; } // 三子推理 InferenceResult tertiary_inference(const
BoardState& state, int depth = 3) { vector<string> legal_moves =
generate_legal_moves(state); if (legal_moves.empty()) { return {"", 0.0, {}, {},
depth, "No legal moves"}; } vector<float> scores; float best_score = -1e9; string
best_move = legal_moves[0]; // 遍历合法走法 for (const string& move : legal_moves)
{ BoardState new_state = make_move(state, move); float score =
minimax(new_state, depth - 1, false, -1e9, 1e9); scores.push_back(score); if
((state.white_to_move && score > best_score) || (!state.white_to_move && score
< best_score)) { best_score = score; best_move = move; } } // 计算置信度 float
confidence = calculate_confidence(scores, best_score); return { best_move,
confidence, legal_moves, scores, depth, "Minimax search depth=" +
to_string(depth) }; } // 极大极小值 float minimax(BoardState state, int depth, bool
maximizing_player, float alpha, float beta) { if (depth == 0 ||
is_game_over(state)) { return evaluate_position(state); } vector<string> moves
= generate_legal_moves(state); if (moves.empty()) { return maximizing_player ?
-1e9 : 1e9; // 边界值 } if (maximizing_player) { float max_eval = -1e9; for (const
string& move : moves) { BoardState new_state = make_move(state, move); float
eval = minimax(new_state, depth - 1, false, alpha, beta); max_eval =
max(max_eval, eval); alpha = max(alpha, eval); if (beta <= alpha) break; //
Alpha-Beta 剪枝 } return max_eval; } else { float min_eval = 1e9; for (const
string& move : moves) { BoardState new_state = make_move(state, move); float

```



```

eval = minimax(new_state, depth - 1, true, alpha, beta); min_eval =
min(min_eval, eval); beta = min(beta, eval); if (beta <= alpha) break; // Alpha-
Beta } return min_eval; } } // float evaluate_position(const
BoardState& state) { float material = 0.0; float mobility = 0.0; float king_safety
= 0.0; float pawn_structure = 0.0; // for (const auto& row : state.board)
{ for (char piece : row) { if (piece != '.') { material += piece_values[piece]; } } }
// int white_moves = count_legal_moves(state, true); int black_moves =
count_legal_moves(state, false); mobility = (white_moves - black_moves) * 0.1; //
king_safety = evaluate_king_safety(state); // pawn_structure =
evaluate_pawn_structure(state); return material + mobility + king_safety +
pawn_structure; } // vector<string> generate_legal_moves(const
BoardState& state) { vector<string> moves; // for (int i = 0; i < 8; ++i) { for (int j = 0; j < 8; ++j) { char piece = state.board[i][j]; if
((state.white_to_move && piece.isupper()) || (!state.white_to_move &&
piece.islower())) { // string from = string(1, 'a' + j) + to_string(8 - i);
string to = string(1, 'a' + (j + 1) % 8) + to_string(8 - (i + 1) % 8);
moves.push_back(from + to); } } } return moves; } // float
detect_forks(const BoardState& state) { // int fork_count = 0; for (int i = 0; i < 8; ++i) { for (int j = 0; j < 8; ++j) { char piece = state.board[i][j]; if
(toupper(piece) == 'N' || toupper(piece) == 'Q') { // int
attack_count = 0; for (int di = -1; di <= 1; ++di) { for (int dj = -1; dj <= 1; ++dj)
{ if (di == 0 && dj == 0) continue; int ni = i + di * 2, nj = j + dj * 2; if (0 <= ni <
8 && 0 <= nj < 8 && state.board[ni][nj] != '.') { attack_count++; } ni = i + di *
1, nj = j + dj * 1; if (0 <= ni < 8 && 0 <= nj < 8 && state.board[ni][nj] != '.')
{ attack_count++; } } } if (attack_count >= 2) { fork_count++; } } } } return
fork_count * 0.5; } float detect_pins(const BoardState& state) { // int
pin_count = 0; // return pin_count * 0.3; } float
detect_checkmate(const BoardState& state) { // vector<string> moves
= generate_legal_moves(state); if (moves.empty() && is_in_check(state))
{ return 10.0; // } return 0.0; } float evaluate_king_safety(const
BoardState& state) { // float white_safety = 0.0, black_safety = 0.0; //
for (int i = 0; i < 8; ++i) { for (int j = 0; j < 8; ++j) { if
(state.board[i][j] == 'K') { white_safety = count_defenders(state, i, j, true); }
else if (state.board[i][j] == 'k') { black_safety = count_defenders(state, i, j,
false); } } } return state.white_to_move ? white_safety : -black_safety; } float
count_defenders(const BoardState& state, int x, int y, bool is_white) { int count
= 0; for (int di = -1; di <= 1; ++di) { for (int dj = -1; dj <= 1; ++dj) { if (di == 0
&& dj == 0) continue; int ni = x + di, nj = y + dj; if (0 <= ni < 8 && 0 <= nj < 8)
{ char piece = state.board[ni][nj]; if ((is_white && piece.isupper()) || (!is_white
&& piece.islower())) { count++; } } } } return count * 0.2; } float
evaluate_pawn_structure(const BoardState& state) { // float
doubled_pawns = 0.0; float isolated_pawns = 0.0; for (int j = 0; j < 8; ++j) { int
pawn_count = 0; for (int i = 0; i < 8; ++i) { if (toupper(state.board[i][j]) == 'P') {
pawn_count++; } } if (pawn_count >= 2) { doubled_pawns -= (pawn_count - 1)
* 0.5; } } return doubled_pawns + isolated_pawns; } bool is_game_over(const
BoardState& state) { // return
generate_legal_moves(state).empty(); } bool is_in_check(const BoardState&

```

```

state) { // 返回 false; } BoardState make_move(const BoardState&
state, const string& move) { // BoardState new_state = state; int from_x
= move[0] - 'a'; int from_y = 8 - (move[1] - '0'); int to_x = move[2] - 'a'; int to_y
= 8 - (move[3] - '0'); new_state.board[to_y][to_x] = new_state.board[from_y]
[from_x]; new_state.board[from_y][from_x] = '.'; new_state.white_to_move = !
new_state.white_to_move; new_state.move_number++; return new_state; } int
count_legal_moves(const BoardState& state, bool white_to_move) { // BoardState
BoardState temp = state; temp.white_to_move = white_to_move; return
generate_legal_moves(temp).size(); } float calculate_confidence(const
vector<float>& scores, float best_score) { // if (scores.empty())
return 0.0; float score_range = *max_element(scores.begin(), scores.end()) -
*min_element(scores.begin(), scores.end()); if (score_range < 0.1) { return 0.3; //
} float margin = 0.0; for (float score : scores) { if (abs(score -
best_score) > 0.01) { margin = max(margin, abs(best_score - score)); } } return
min(1.0f, margin / score_range); } public: ChessInferenceEngine() :
rng(random_device{}()) {} // json run_inference(const json&
state_json, int inference_level = 3) { BoardState state; // JSON
vector<vector<char>> board(8, vector<char>(8, '.')); string fen =
state_json["fen"]; parse_fen(fen, board); state.board = board;
state.white_to_move = state_json["white_to_move"]; state.move_number =
state_json["move_number"]; state.evaluation = state_json.value("evaluation",
0.0f); // if (inference_level == 1) { float eval =
primary_inference(state); return { {"type", "primary_inference"}, {"evaluation",
eval}, {"confidence", 0.7f}, {"depth", 1} }; } else if (inference_level == 2)
{ auto tactics = secondary_inference(state); json tactics_json; for (const auto&
tactic : tactics) { tactics_json[tactic.first] = tactic.second; } return { {"type",
"secondary_inference"}, {"tactics", tactics_json}, {"confidence", 0.85f},
{"depth", 2} }; } else { // Level 3 InferenceResult result =
tertiary_inference(state, 3); return { {"type", "tertiary_inference"},
{"best_move", result.best_move}, {"confidence", result.confidence},
{"candidate_moves", result.candidate_moves}, {"move_scores",
result.move_scores}, {"depth", result.inference_depth}, {"reasoning",
result.reasoning_path} }; } } // FEN void parse_fen(const string& fen,
vector<vector<char>>& board) { string board_part = fen.substr(0, fen.find(' '));
int row = 0; size_t pos = 0; while (pos < board_part.size()) { char c =
board_part[pos]; if (c == '/') { row++; pos++; } else if (isdigit(c)) { int count = c
- '0'; for (int i = 0; i < count; ++i) { board[row][pos - row - i] = '.'; } pos++; }
else { board[row][pos - row] = c; pos++; } } } // int main()
{ ChessInferenceEngine engine; // FEN json state = { {"fen",
"rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"},
{"white_to_move", true}, {"move_number", 1}, {"evaluation", 0.0} }; //
json result = engine.run_inference(state, 3); cout << "FEN:" << endl; cout <<
result.dump(4) << endl; return 0; } ```` 2. ````cpp#include
<vector>#include <random>#include <cmath>#include <iostream>#include
<nlohmann/json.hpp>using json = nlohmann::json;using namespace std; //
class BioNeuron {private: double membrane_potential_; // double
resting_potential_; // (-70mV) double threshold_; // (-55mV) double

```

```

refractory_period_; // 0.0 double refractory_timer_; // 0.0 vector<double>
synapses_; // 0.0 mt19937 rng_; // 0.0 const double g_Na = 120.0; // 0.0
const double g_K = 36.0; // 0.0 const double g_L = 0.3; // 0.0 const
double E_Na = 50.0; // 0.0 const double E_K = -77.0; // 0.0 const double E_L
= -54.387; // 0.0 // Hodgkin-Huxley void update_hh_model(double
input_current, double dt) { if (refractory_timer_ > 0) { refractory_timer_ -= dt;
membrane_potential_ = resting_potential_; return; } double V =
membrane_potential_; // 0.0 double m = 1.0 / (1.0 + exp((-V - 40.0)/10.0));
double h = 1.0 / (1.0 + exp((V + 65.0)/18.0)); double n = 1.0 / (1.0 + exp((-V -
55.0)/10.0)); // 0.0 double I_Na = g_Na * pow(m, 3) * h * (V - E_Na); double
I_K = g_K * pow(n, 4) * (V - E_K); double I_L = g_L * (V - E_L); // 0.0 double
dV_dt = (input_current - I_Na - I_K - I_L) / 1.0; // 0.0 C=1.0 membrane_potential_
+= dV_dt * dt; // 0.0 if (membrane_potential_ > 0.0) { // 0.0 0mV
membrane_potential_ = 30.0; // 0.0 refractory_timer_ =
refractory_period_; } // 0.0 membrane_potential_ = max(-85.0,
min(membrane_potential_, 40.0)); } public: BioNeuron(int input_size, double
resting_potential = -70.0, double threshold = -55.0) :
membrane_potential_(resting_potential), resting_potential_(resting_potential),
threshold_(threshold), refractory_period_(5.0), // 5ms refractory_timer_(0.0),
rng_(random_device{ }()) { // 0.0 normal_distribution<double>
dist(0.0, 0.1); synapses_.reserve(input_size); for (int i = 0; i < input_size; ++i)
{ synapses_.push_back(dist(rng_)); } } // 0.0 bool activate(const
vector<double>& inputs, double dt = 0.1) { if (inputs.size() != synapses_.size())
{ throw invalid_argument(" "); } // 0.0 double input_current
= 0.0; for (size_t i = 0; i < inputs.size(); ++i) { input_current += inputs[i] *
synapses_[i]; } // 0.0 Hodgkin-Huxley update_hh_model(input_current, dt); // 0.0
bool fired = (membrane_potential_ > 0.0); if (fired) { refractory_timer_
= refractory_period_; } return fired; } // 0.0 void
hebbian_learning(const vector<double>& inputs, double learning_rate = 0.01)
{ if (refractory_timer_ > 0) return; // 0.0 double activity =
(membrane_potential_ - resting_potential_) / (threshold_ - resting_potential_);
activity = max(0.0, min(1.0, activity)); // 0.0 0-1 for (size_t i = 0; i <
synapses_.size(); ++i) { synapses_[i] += learning_rate * inputs[i] * activity;
synapses_[i] = max(-2.0, min(synapses_[i], 2.0)); // 0.0 } } // STDP
void stdp_learning(double pre_spike_time, double post_spike_time, double
dt = 0.1) { double delta_t = post_spike_time - pre_spike_time; double
learning_rate = delta_t > 0 ? 0.001 : -0.0005; if (abs(delta_t) < 20.0) { // 0.0 20ms
double weight_change = learning_rate * exp(-abs(delta_t)/10.0); for
(auto& synapse : synapses_) { synapse += weight_change; synapse = max(-2.0,
min(synapse, 2.0)); } } } double get_membrane_potential() const { return
membrane_potential_; } const vector<double>& get_synapses() const { return
synapses_; } }; // 0.0 class BioNeuralNetwork { private:
vector<vector<BioNeuron>> layers_; int input_size_; vector<int> hidden_sizes_;
int output_size_; mt19937 rng_; public: BioNeuralNetwork(int input_size, const
vector<int>& hidden_sizes, int output_size) : input_size_(input_size),
hidden_sizes_(hidden_sizes), output_size_(output_size), rng_(random_device{ }())
{ // 0.0 if (!hidden_sizes.empty()) { layers_.emplace_back(hidden_sizes[0],

```

```

BioNeuron(input_size)); for (size_t i = 1; i < hidden_sizes.size(); ++i)
{ layers_.emplace_back(hidden_sizes[i], BioNeuron(hidden_sizes[i-1])); }
layers_.emplace_back(output_size, BioNeuron(hidden_sizes.back())); } else
{ layers_.emplace_back(output_size, BioNeuron(input_size)); } } // 0000
vector<bool> forward(const vector<double>& inputs, double dt = 0.1) { if
(inputs.size() != input_size_) { throw invalid_argument("00000000"); }
vector<double> current_inputs = inputs; vector<bool> output_spikes; // 0000
for (auto& layer : layers_) { vector<double> next_inputs(layer.size(), 0.0);
vector<bool> spikes(layer.size(), false); for (size_t i = 0; i < layer.size(); ++i)
{ spikes[i] = layer[i].activate(current_inputs, dt); next_inputs[i] =
layer[i].get_membrane_potential(); } current_inputs = next_inputs; if (&layer ==
&layers_.back()) { output_spikes = spikes; } } return output_spikes; } // 000000
void backward(const vector<double>& inputs, const vector<bool>& target,
double learning_rate = 0.01, double dt = 0.1) { // 000000000000
vector<vector<double>> layer_potentials; vector<double> current_inputs =
inputs; for (auto& layer : layers_) { vector<double> potentials(layer.size(), 0.0);
for (size_t i = 0; i < layer.size(); ++i) { layer[i].activate(current_inputs, dt);
potentials[i] = layer[i].get_membrane_potential(); }
layer_potentials.push_back(potentials); current_inputs = potentials; } // 000000
auto& output_layer = layers_.back(); for (size_t i = 0; i < output_layer.size(); +
+i) { bool fired = (layer_potentials.back()[i] > 0); double error = (target[i] ? 1.0 :
0.0) - (fired ? 1.0 : 0.0); if (abs(error) > 0.1)
{ output_layer[i].hebbian_learning(layer_potentials[layers_.size()-2],
learning_rate * abs(error)); } } // 00000000000000 for (int l = layers_.size()-2; l >= 0;
--l) { auto& layer = layers_[l]; const auto& next_layer = layers_[l+1]; for (size_t i
= 0; i < layer.size(); ++i) { double error = 0.0; for (size_t j = 0; j <
next_layer.size(); ++j) { error += next_layer[j].get_synapses()[i]; }
layer[i].hebbian_learning(l == 0 ? inputs : layer_potentials[l-1], learning_rate *
abs(error) * 0.1); } } } // 00000000 json export_config() const { json config;
config["input_size"] = input_size_; config["hidden_sizes"] = hidden_sizes_;
config["output_size"] = output_size_; json layers; for (const auto& layer : layers_)
{ json layer_json; for (const auto& neuron : layer)
{ layer_json.push_back({ {"synapses", neuron.get_synapses()},
{"membrane_potential", neuron.get_membrane_potential()} }); }
layers.push_back(layer_json); } config["layers"] = layers; return config; } } // 0000
0000000000 class ChessNeuralProcessor {private: BioNeuralNetwork nn_; mt19937
rng_;public: ChessNeuralProcessor() : nn_(64, {256, 128, 64}, 4096),
rng_(random_device{ }()) { // 64 0000000000003 0000004096 000000000000 } // 00000000
json evaluate_position(const json& board_state) { vector<double> inputs =
convert_board_to_input(board_state); vector<bool> spikes =
nn_.forward(inputs); // 000000000000 vector<double> move_probabilities; for (bool
spike : spikes) { move_probabilities.push_back(spike ? 1.0 : 0.0); } // 000000
double sum = 0.0; for (double p : move_probabilities) sum += p; if (sum > 0)
{ for (double& p : move_probabilities) p /= sum; } // 00000000 int best_move_idx =
distance(move_probabilities.begin(), max_element(move_probabilities.begin(),
move_probabilities.end())); string best_move = index_to_move(best_move_idx);
return { {"best_move", best_move}, {"move_probabilities", move_probabilities},

```

```

{"confidence", move_probabilities[best_move_idx]}, {"evaluation",
calculate_evaluation(move_probabilities)} } }private: vector<double>
convert_board_to_input(const json& board_state) { vector<double> inputs(64,
0.0); vector<vector<char>> board = board_state["board"]; // 棋盘状态
const map<char, double> piece_values = { {'P', 1.0}, {'N', 3.2}, {'B', 3.3}, {'R',
5.0}, {'Q', 9.0}, {'K', 100.0}, {'p', -1.0}, {'n', -3.2}, {'b', -3.3}, {'r', -5.0}, {'q', -
9.0}, {'k', -100.0} }; for (int i = 0; i < 8; ++i) { for (int j = 0; j < 8; ++j) { char
piece = board[i][j]; inputs[i*8 + j] = piece_values.count(piece) ?
piece_values.at(piece) : 0.0; } } return inputs; } string index_to_move(int index)
{ // 将索引转换为 UCI 格式 int from = index / 64; int to = index % 64; char from_col =
'a' + (from % 8); int from_row = 8 - (from / 8); char to_col = 'a' + (to % 8); int
to_row = 8 - (to / 8); return string(1, from_col) + to_string(from_row) + string(1,
to_col) + to_string(to_row); } double calculate_evaluation(const
vector<double>& move_probabilities) { // 计算评估值 double max_prob =
*max_element(move_probabilities.begin(), move_probabilities.end()); double
entropy = 0.0; for (double p : move_probabilities) { if (p > 0) { entropy -= p *
log2(p); } } // 信息熵 = 最大概率 - 熵 return max_prob - entropy /
log2(move_probabilities.size()); } };// 主函数 int main() { ChessNeuralProcessor
processor; // 棋盘状态 json board_state = { {"board",
vector<vector<char>>{ {'r','n','b','q','k','b','n','r'}, {'p','p','p','p','p','p','p','p'},
{'.....'}, {'.....'}, {'.....'},
{'.....'}, {'P','P','P','P','P','P','P','P'}, {'R','N','B','Q','K','B','N','R'} }},
{"white_to_move", true}, {"move_number", 1} }; json result =
processor.evaluate_position(board_state); cout << "结果:" << endl; cout
<< result.dump(4) << endl; return 0;} ``---## Java 实现 ##
1. 使用 Java 实现
javapackage com.chess.consciousness;import
com.google.gson.Gson;import com.google.gson.JsonObject;import
java.util.*;import java.util.concurrent.*;import java.util.stream.Collectors;// 定义
AgentComponent 接口
interface AgentComponent { String getId(); ComponentType getType();
JsonObject process(JsonObject input); double getConfidence(); void
updateParameters(Map<String, Object> params);}// 定义 ComponentType 枚举
enum ComponentType { PERCEPTION, MEMORY, INFERENCE, DECISION, EVALUATION,
LEARNING}// 实现 ChessConsciousnessAgent 类
class ChessConsciousnessAgent { private String agentId;
private Map<ComponentType, List<AgentComponent>> components; private
ExecutorService executor; private ScheduledExecutorService scheduler; private
Gson gson; private JsonObject currentState; private List<JsonObject>
memoryTraces; public ChessConsciousnessAgent() { this.agentId =
UUID.randomUUID().toString(); this.components = new
EnumMap<>(ComponentType.class); this.executor =
Executors.newFixedThreadPool(10); this.scheduler =
Executors.newScheduledThreadPool(2); this.gson = new Gson();
this.memoryTraces = new ArrayList<>(); // 初始化组件
private void initializeComponents() { // 添加感知组件
components.put(ComponentType.PERCEPTION, Arrays.asList(new
VisionPerceptionComponent(), new LanguageProcessingComponent())); // 添加记忆组件
components.put(ComponentType.MEMORY,
Arrays.asList(shortTermMemoryComponent(), longTermMemoryComponent(),

```

```

associativeMemoryComponent())); // []
components.put(ComponentType.INFERENCE, Arrays.asList(new
PrimaryInferenceComponent(), new SecondaryInferenceComponent(), new
TertiaryInferenceComponent())); // []
components.put(ComponentType.DECISION, Collections.singletonList(new
DecisionMakingComponent())); // []
components.put(ComponentType.EVALUATION, Arrays.asList(new
SelfEvaluationComponent(), new SystemEvaluationComponent())); } // []
public JsonObject processTurn(JsonObject sensoryInput) { long startTime =
System.currentTimeMillis(); // 1. [] JsonObject perceptionResult =
processComponent(ComponentType.PERCEPTION, sensoryInput); currentState =
perceptionResult.deepCopy(); // 2. [] JsonObject memoryResult =
processComponent(ComponentType.MEMORY, currentState);
currentState.add("memory_data", memoryResult); // 3. [] JsonObject
inferenceResult = processComponent(ComponentType.INFERENCE,
currentState); currentState.add("inference_data", inferenceResult); // 4. []
JsonObject decisionResult = processComponent(ComponentType.DECISION,
currentState); currentState.add("decision", decisionResult); // 5. []
JsonObject evaluationResult = processComponent(ComponentType.EVALUATION,
currentState); currentState.add("evaluation", evaluationResult); // 6. []
updateMemory(currentState); // []
currentState.addProperty("processing_time_ms", System.currentTimeMillis() -
startTime); currentState.addProperty("agent_id", agentId); return currentState; }
// [] private JsonObject processComponent(ComponentType type,
JsonObject input) { List<AgentComponent> componentList =
components.getDefault(type, Collections.emptyList()); if
(componentList.isEmpty()) { return new JsonObject(); } // []
List<CompletableFuture<JsonObject>> futures =
componentList.stream().map(comp -> CompletableFuture.supplyAsync(() ->
comp.process(input), executor)).collect(Collectors.toList()); // []
CompletableFuture<Void> allOf = CompletableFuture.allOf(futures.toArray(new
CompletableFuture[0])); try { allOf.get(30, TimeUnit.SECONDS); // 30 []
List<JsonObject> results =
futures.stream().map(CompletableFuture::join).collect(Collectors.toList());
return fuseComponentResults(type, results, input); } catch (InterruptedException
| ExecutionException | TimeoutException e) { JsonObject error = new
JsonObject(); error.addProperty("error", "Component processing failed: " +
e.getMessage()); error.addProperty("component_type", type.name()); return
error; } } // [] private JsonObject fuseComponentResults(ComponentType
type, List<JsonObject> results, JsonObject input) { JsonObject fused = new
JsonObject(); if (type == ComponentType.INFERENCE) { // [] double
totalConfidence = 0.0; double weightedEvaluation = 0.0; JsonObject bestMove =
new JsonObject(); for (JsonObject result : results) { double confidence =
result.get("confidence").getAsDouble(); totalConfidence += confidence; if
(result.has("evaluation")) { weightedEvaluation +=
result.get("evaluation").getAsDouble() * confidence; } if (result.has("best_move")
&& (!bestMove.has("confidence") || confidence >

```

```

bestMove.get("confidence").getAsDouble())) { bestMove = result; } }
fused.add("best_move", bestMove.get("best_move"));
fused.addProperty("evaluation", weightedEvaluation / totalConfidence);
fused.addProperty("average_confidence", totalConfidence / results.size());
fused.add("all_results", gson.toJsonTree(results)); } else { // [] for
(JsonObject result : results) { for (String key : result.keySet()) { if (!
fused.has(key)) { fused.add(key, result.get(key)); } else { // [] if
(result.has("confidence") && fused.has("confidence")) { double resultConf =
result.get("confidence").getAsDouble(); double fusedConf =
fused.get("confidence").getAsDouble(); if (resultConf > fusedConf)
{ fused.add(key, result.get(key)); } } } } } return fused; } // [] private void
updateMemory(JsonObject state) { // []
memoryTraces.add(state.deepCopy()); if (memoryTraces.size() > 100)
{ memoryTraces.remove(0); // [] 100 [] } // []
scheduler.scheduleAtFixedRate(() -> { if (memoryTraces.size() > 10)
{ saveLongTermMemory(memoryTraces.subList(0, 10));
memoryTraces.subList(0, 10).clear(); } }, 5, 60, TimeUnit.SECONDS); } private
void saveLongTermMemory(List<JsonObject> memories) { // []
System.out.println("Saving " + memories.size() + " memory traces to
database"); } // [] private AgentComponent shortTermMemoryComponent()
{ return new AgentComponent() { private final String id = "STM-" +
UUID.randomUUID(); private final int maxCapacity = 100; private final
Deque<JsonObject> memory = new LinkedList<>(); @Override public String
getId() { return id; } @Override public ComponentType getType() { return
ComponentType.MEMORY; } @Override public JsonObject process(JsonObject
input) { // [] memory.addFirst(input.deepCopy()); if (memory.size() >
maxCapacity) { memory.removeLast(); } // [] List<JsonObject>
similarStates = findSimilarStates(input); JsonObject result = new JsonObject();
result.addProperty("memory_size", memory.size()); result.add("similar_states",
gson.toJsonTree(similarStates)); result.addProperty("confidence",
calculateConfidence(similarStates.size())); return result; } @Override public
double getConfidence() { return 0.85; } @Override public void
updateParameters(Map<String, Object> params) { // [] } private
List<JsonObject> findSimilarStates(JsonObject target) { List<JsonObject> similar
= new ArrayList<>(); String targetFen = target.get("fen").getString(); for
(JsonObject state : memory) { if (state.has("fen")) { String fen =
state.get("fen").getString(); if (calculateFenSimilarity(targetFen, fen) > 0.8)
{ similar.add(state); } } } return
similar.stream().sorted(Comparator.comparingDouble(s -> -
calculateFenSimilarity(targetFen,
s.get("fen").getString()))).limit(5).collect(Collectors.toList()); } private double
calculateFenSimilarity(String fen1, String fen2) { // [] FEN [] String[] parts1
= fen1.split(" ")[0].split("/"); String[] parts2 = fen2.split(" ")[0].split("/"); int
matches = 0; int total = 0; for (int i = 0; i < 8; i++) { for (int j = 0; j < 8; j++)
{ char c1 = parts1[i].charAt(j); char c2 = parts2[i].charAt(j); if
(Character.isLetter(c1) && Character.isLetter(c2)) { total++; if
(Character.toUpperCase(c1) == Character.toUpperCase(c2)) { matches++; } } } } }

```

```

 } return total > 0 ? (double) matches / total : 0.0; } private double
 calculateConfidence(int similarCount) { return Math.min(1.0, 0.5 + similarCount
 * 0.1); } }; } private AgentComponent longTermMemoryComponent() { // 长期记忆组件
 return new AgentComponent() { @Override public String getId() { return
 "LTM-" + UUID.randomUUID(); } @Override public ComponentType getType()
 { return ComponentType.MEMORY; } @Override public JsonObject
 process(JsonObject input) { // 长期记忆检索 JsonObject result = new JsonObject();
 result.addProperty("message", "Long-term memory retrieval not implemented");
 result.addProperty("confidence", 0.7); return result; } @Override public double
 getConfidence() { return 0.7; } @Override public void
 updateParameters(Map<String, Object> params) {} }; } private
 AgentComponent associativeMemoryComponent() { // 关联记忆组件 return new
 AgentComponent() { @Override public String getId() { return "AM-" +
 UUID.randomUUID(); } @Override public ComponentType getType() { return
 ComponentType.MEMORY; } @Override public JsonObject process(JsonObject
 input) { JsonObject result = new JsonObject(); result.add("associated_patterns",
 gson.toJsonTree(findAssociatedPatterns(input)));
 result.addProperty("confidence", 0.8); return result; } @Override public double
 getConfidence() { return 0.8; } @Override public void
 updateParameters(Map<String, Object> params) {} private List<String>
 findAssociatedPatterns(JsonObject input) { List<String> patterns = new
 ArrayList<>(); // 关联模式 patterns.add("king_safety_pattern");
 patterns.add("center_control_pattern"); patterns.add("development_pattern");
 return patterns; } }; } // 视觉感知组件 private static class
 VisionPerceptionComponent implements AgentComponent { @Override public
 String getId() { return "VISION-" + UUID.randomUUID(); } @Override public
 ComponentType getType() { return ComponentType.PERCEPTION; } @Override
 public JsonObject process(JsonObject input) { JsonObject result = new
 JsonObject(); result.addProperty("message", "Vision processing completed");
 result.addProperty("confidence", 0.92); return result; } @Override public double
 getConfidence() { return 0.92; } @Override public void
 updateParameters(Map<String, Object> params) {} } private static class
 LanguageProcessingComponent implements AgentComponent { @Override
 public String getId() { return "LANG-" + UUID.randomUUID(); } @Override public
 ComponentType getType() { return ComponentType.PERCEPTION; } @Override
 public JsonObject process(JsonObject input) { JsonObject result = new
 JsonObject(); result.addProperty("message", "Language processing completed");
 result.addProperty("confidence", 0.88); return result; } @Override public double
 getConfidence() { return 0.88; } @Override public void
 updateParameters(Map<String, Object> params) {} } // 初级推理组件
 PrimaryInferenceComponent 初级推理... // 初级推理 private JsonObject
 deepCopy(JsonObject json) { return gson.fromJson(gson.toJson(json),
 JsonObject.class); } } // 主类 public class ChessConsciousnessMain { public static
 void main(String[] args) { ChessConsciousnessAgent agent = new
 ChessConsciousnessAgent(); // 初始输入 JsonObject sensoryInput = new
 JsonObject(); sensoryInput.addProperty("fen",
 "rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1");

```



```

sensoryInput.addProperty("visual_confidence", 0.95);
sensoryInput.addProperty("audio_input", "White to move"); // 初始化 JSONObject
result = agent.processTurn(sensoryInput); System.out.println("棋局结果:");
System.out.println(result); }} ``---## 在MATLAB 中实现蒙特卡罗树搜索 1. 初始化
``matlabclassdef ChessMathOptimizer < handle properties (Access = private)
rngStream pieceValues positionTables end methods function obj =
ChessMathOptimizer() % 初始化 obj rngStream = RandStream('mt19937ar', 'Seed',
sum(clock)); obj.pieceValues = struct(... 'P', 1.0, 'N', 3.2, 'B', 3.3, 'R', 5.0, 'Q', 9.0,
'K', 100.0, ... 'p', -1.0, 'n', -3.2, 'b', -3.3, 'r', -5.0, 'q', -9.0, 'k', -100.0); % 初始化
obj.positionTables = obj.initializePositionTables(); end function score =
evaluatePosition(obj, boardState) % 计算材料得分 materialScore =
obj.calculateMaterialScore(boardState); positionalScore =
obj.calculatePositionalScore(boardState); tacticalScore =
obj.calculateTacticalScore(boardState); kingSafetyScore =
obj.calculateKingSafety(boardState); % 计算总分 score = materialScore * 1.0 + ...
positionalScore * 0.15 + ... tacticalScore * 0.3 + ... kingSafetyScore * 0.2; end
function move = findBestMove(obj, boardState, depth) % 蒙特卡罗树搜索 root
= obj.createMCTSNode(boardState); % 蒙特卡罗树搜索 for i = 1:1000 selectedNode
= obj.selectNode(root); expandedNode = obj.expandNode(selectedNode);
simulationResult = obj.simulation(expandedNode);
obj.backpropagate(expandedNode, simulationResult); end % 蒙特卡罗树搜索
bestChild = obj.selectBestChild(root); move = bestChild.move; end function
[bestMove, confidence] = optimizeWithCalculus(obj, boardState) % 蒙特卡罗树搜索
legalMoves = obj.generateLegalMoves(boardState); moveScores =
zeros(size(legalMoves)); % 蒙特卡罗树搜索 for i = 1:length(legalMoves) newState =
obj.makeMove(boardState, legalMoves{i}); moveScores(i) =
obj.evaluatePosition(newState); % 蒙特卡罗树搜索 gradient =
obj.calculateScoreGradient(newState); moveScores(i) = moveScores(i) +
norm(gradient) * 0.01; end % 蒙特卡罗树搜索 [maxScore, idx] = max(moveScores);
bestMove = legalMoves{idx}; % 蒙特卡罗树搜索 scoreRange = max(moveScores) -
min(moveScores); if scoreRange < 0.1 confidence = 0.3; else margin =
maxScore - sort(moveScores, 'descend')(2); confidence = min(1, margin /
scoreRange); end end function [eigenvalues, eigenvectors] =
analyzeBoardSymmetry(obj, boardState) % 蒙特卡罗树搜索 boardMatrix =
obj.convertBoardToMatrix(boardState); % 蒙特卡罗树搜索 symmetries =
obj.calculateSymmetries(boardMatrix); % 蒙特卡罗树搜索 [eigenvectors, eigenvalues] =
eig(symmetries); % 蒙特卡罗树搜索 [~, sortIdx] = sort(diag(eigenvalues), 'descend');
eigenvalues = eigenvalues(sortIdx, sortIdx); eigenvectors = eigenvectors(:,
sortIdx); end function complexValue = evaluateComplexPosition(obj, boardState)
% 蒙特卡罗树搜索 material = obj.calculateMaterialScore(boardState); position =
obj.calculatePositionalScore(boardState); % 蒙特卡罗树搜索 complexValue =
complex(material, position); % 蒙特卡罗树搜索 complexValue = log(complexValue +
1i); complexValue = sin(complexValue); end function functional =
calculateFunctional(obj, boardState) % 蒙特卡罗树搜索 boardVector =
obj.convertBoardToVector(boardState); % 蒙特卡罗树搜索 A =
obj.createLinearOperator(size(boardVector, 1)); % 蒙特卡罗树搜索 functional =
boardVector' * A * boardVector + norm(boardVector); end end methods (Access

```

```

= private) function tables = initializePositionTables(obj) % ██████████ tables.pawn
= [... 0, 0, 0, 0, 0, 0, 0, 0; ... 5, 5, 5, 5, 5, 5, 5, 5; ... 1, 1, 2, 3, 3, 2, 1, 1; ... 0.5,
0.5, 1, 2.5, 2.5, 1, 0.5, 0.5; ... 0, 0, 0, 2, 2, 0, 0, 0; ... 0.5, -0.5, -1, 0, 0, -1, -0.5,
0.5; ... 0.5, 1, 1, -2, -2, 1, 1, 0.5; ... 0, 0, 0, 0, 0, 0, 0, 0]; tables.knight = [... -5, -
4, -3, -3, -3, -3, -4, -5; ... -4, -2, 0, 0, 0, 0, -2, -4; ... -3, 0, 1, 1.5, 1.5, 1, 0, -3; ... -3,
0.5, 1.5, 2, 2, 1.5, 0.5, -3; ... -3, 0, 1.5, 2, 2, 1.5, 0, -3; ... -3, 0.5, 1, 1.5, 1.5, 1,
0.5, -3; ... -4, -2, 0, 0.5, 0.5, 0, -2, -4; ... -5, -4, -3, -3, -3, -3, -4, -5]; end function
score = calculateMaterialScore(obj, boardState) % ██████████ score = 0; for i =
1:8 for j = 1:8 piece = boardState.board(i, j); if piece ~= '.' score = score +
obj.pieceValues.(piece); end end end end function score =
calculatePositionalScore(obj, boardState) % ██████████ score = 0; for i = 1:8 for j
= 1:8 piece = boardState.board(i, j); if piece ~= '.' if upper(piece) == 'P' value =
obj.positionTables.pawn(i, j); score = score + (piece == 'P' ? value : -value);
elseif upper(piece) == 'N' value = obj.positionTables.knight(i, j); score = score +
(piece == 'N' ? value : -value); end end end end end function score =
calculateTacticalScore(obj, boardState) % ██████████ score = 0; % ██████████
forkCount = obj.detectForks(boardState); pinCount = obj.detectPins(boardState);
checkmateThreat = obj.detectCheckmate(boardState); score = score +
forkCount * 2.0; score = score + pinCount * 1.5; score = score +
checkmateThreat * 10.0; end function score = calculateKingSafety(obj,
boardState) % ██████████ whiteKingPos = obj.findKing(boardState, true);
blackKingPos = obj.findKing(boardState, false); whiteSafety =
obj.evaluateKingPosition(boardState, whiteKingPos, true); blackSafety =
obj.evaluateKingPosition(boardState, blackKingPos, false); score =
boardState.whiteToMove ? whiteSafety - blackSafety : blackSafety - whiteSafety;
end function gradient = calculateScoreGradient(obj, boardState) % ██████████
gradient = zeros(64, 1); baseScore = obj.evaluatePosition(boardState); % ██████████
██████ for i = 1:8 for j = 1:8 modifiedState = boardState; originalPiece =
modifiedState.board(i, j); % ████████ if originalPiece ~= '.' modifiedState.board(i, j)
= '.'; perturbedScore = obj.evaluatePosition(modifiedState); gradient((i-1)*8 + j)
= baseScore - perturbedScore; modifiedState.board(i, j) = originalPiece; end end
end end function operator = createLinearOperator(obj, size) % ██████████ operator
= sparse(size, size); % ████████████████████████ for i = 1:size operator(i, i) = 4; if i > 1
operator(i, i-1) = -1; end if i < size operator(i, i+1) = -1; end if i > 8 operator(i, i-
8) = -1; end if i <= size-8 operator(i, i+8) = -1; end end end % ████████████████████████
function node = createMCTSNode(obj, state) node.state = state; node.visitCount
= 0; node.totalReward = 0; node.children = []; node.move = []; end function
node = selectNode(obj, node) % UCB1 █████ while ~isempty(node.children)
ucbValues = zeros(size(node.children)); for i = 1:length(node.children) child =
node.children{i}; exploitation = child.totalReward / child.visitCount; exploration
= sqrt(2 * log(node.visitCount) / child.visitCount); ucbValues(i) = exploitation +
1.414 * exploration; end [~, idx] = max(ucbValues); node = node.children{idx};
end end function node = expandNode(obj, node) % ████████ legalMoves =
obj.generateLegalMoves(node.state); node.children = cell(size(legalMoves)); for i
= 1:length(legalMoves) newState = obj.makeMove(node.state, legalMoves{i});
node.children{i} = obj.createMCTSNode(newState); node.children{i}.move =
legalMoves{i}; end return node.children{1}; % ████████████████████████ end function result =

```

```

simulation(obj, node) % ===== currentState = node.state; moveCount = 0; while
moveCount < 50 legalMoves = obj.generateLegalMoves(currentState); if
isempty(legalMoves) break; end % ===== moveldx = randi(obj.rngStream,
length(legalMoves)); currentState = obj.makeMove(currentState,
legalMoves{moveldx}); moveCount = moveCount + 1; end result =
obj.evaluatePosition(currentState); end function backpropagate(obj, node, result)
% ===== while ~isempty(node) node.visitCount = node.visitCount + 1;
node.totalReward = node.totalReward + result; node = node.parent; % =====
===== end end function child = selectBestChild(obj, node) % =====
visitCounts = cellfun(@(c) c.visitCount, node.children); [~, idx] =
max(visitCounts); child = node.children{idx}; end % ===== function moves =
generateLegalMoves(obj, boardState) % ===== moves = {}; % =====
for i = 1:8 for j = 1:8 piece = boardState.board(i, j); if
(boardState.whiteToMove && isupper(piece)) || ... (~boardState.whiteToMove &&
islower(piece)) from = [i, j]; to = [mod(i, 8)+1, mod(j, 8)+1]; moves{end+1} =
obj.coordsToMove(from, to); end end end end function state = makeMove(obj,
state, move) % ===== state = state; % ===== [from, to] =
obj.moveToCoords(move); state.board(to(1), to(2)) = state.board(from(1),
from(2)); state.board(from(1), from(2)) = '.'; state.whiteToMove =
~state.whiteToMove; state.moveNumber = state.moveNumber + 1; end function
pos = findKing(obj, boardState, isWhite) % ===== kingChar = isWhite ? 'K' : 'k';
for i = 1:8 for j = 1:8 if boardState.board(i, j) == kingChar pos = [i, j]; return;
end end end pos = []; end function score = evaluateKingPosition(obj, boardState,
pos, isWhite) % ===== score = 0; if isempty(pos) return; end % ===== for di
= -1:1 for dj = -1:1 if di == 0 && dj == 0 continue; end ni = pos(1) + di; nj =
pos(2) + dj; if ni < 1 || ni > 8 || nj < 1 || nj > 8 continue; end piece =
boardState.board(ni, nj); if piece == '.' score = score + 0.1; elseif (isWhite &&
isupper(piece)) || (~isWhite && islower(piece)) score = score + 0.2; end end end
end function count = detectForks(obj, boardState) % ===== count = 0; % =====
end function count = detectPins(obj, boardState) % ===== count = 0; % =====
end function threat = detectCheckmate(obj, boardState) % ===== threat = 0; %
===== end function matrix = convertBoardToMatrix(obj, boardState) % =====
matrix = zeros(8, 8); for i = 1:8 for j = 1:8 piece = boardState.board(i, j); if piece
~= '.' matrix(i, j) = obj.pieceValues.(piece); end end end end function vector =
convertBoardToVector(obj, boardState) % ===== vector = zeros(64, 1); for i
= 1:8 for j = 1:8 piece = boardState.board(i, j); if piece ~= '.' vector((i-1)*8 + j)
= obj.pieceValues.(piece); end end end end function symmetries =
calculateSymmetries(obj, boardMatrix) % ===== symmetries = zeros(8, 8); %
===== for i = 1:4 symmetries(i, :) = boardMatrix(i, :) - boardMatrix(9-i, :);
end end function move = coordsToMove(obj, from, to) % ===== fromChar =
[char(96 + from(2)), num2str(9 - from(1))]; toChar = [char(96 + to(2)),
num2str(9 - to(1))]; move = [fromChar, toChar]; end function [from, to] =
moveToCoords(obj, move) % ===== fromCol = move(1) - 96; fromRow = 9 -
str2double(move(2)); toCol = move(3) - 96; toRow = 9 - str2double(move(4));
from = [fromRow, fromCol]; to = [toRow, toCol]; end endend % ===== function
demoChessOptimizer() % ===== optimizer = ChessMathOptimizer(); % =====
boardState = struct(); boardState.board = [... 'r','n','b','q','k','b','n','r'; ...

```

[illegible]



Represent sensory signals as complex functions  $f(z) = u + iv$  .- Integral Transforms: Fourier/Laplace processing for temporal states.- Functional Analysis: Value function  $J: \text{State} \rightarrow \mathbb{R}$  .- Group Theory: Symmetry transformations (e.g.,  $SO(2)$  ).- Graph Theory: State graphs, threat maps, reachability analysis.- Mathematical Logic: First-order predicate calculus.- Visual Thinking: Spatial heatmaps, attention graphs.

## 2. MySQL Database Design

```
sql
CREATE DATABASE chess_consciousness;
USE chess_consciousness;
-- Sensory Memory
CREATE TABLE sensory_memory (
 id INT PRIMARY KEY
 AUTO_INCREMENT,
 step INT,
 visual_state JSON,
 auditory_signal FLOAT,
 tactile_feedback FLOAT,
 olfactory_simulation FLOAT,
 complex_state TEXT
);
-- Memory Levels
CREATE TABLE memory_level (
 id INT PRIMARY KEY
 AUTO_INCREMENT,
 level ENUM('basic','l1','l2','high'),
 content TEXT,
 pattern_type VARCHAR(50),
 embedding BLOB
);
-- Reasoning Logs
CREATE TABLE reasoning_process (
 step INT,
 agent ENUM('A','B'),
 level1 TEXT,
 level2 TEXT,
 level3 TEXT,
 judgment FLOAT,
 decision TEXT,
 self_check INT,
 system_verify INT,
 consciousness_score FLOAT
);
-- Game States
CREATE TABLE chess_game (
 step INT PRIMARY KEY,
 fen VARCHAR(100),
 player CHAR(1),
 move VARCHAR(10),
 value REAL,
 threat_map BLOB
);
```

## 3. MATLAB Mathematical Core

```
matlab
% Complex Sensory Fusion
function z = sensory_complex(visual, auditory, tactile, olfactory)
u = 0.6*visual + 0.1*auditory + 0.2*tactile + 0.1*olfactory;
v = fft(u);
z = u + 1i*v(1:length(u));
end
% Functional Value Evaluation
function J = functional_value(state)
material = sum(state.pieces);
mobility = length(state.legal_moves);
threat = graph centrality(state.threat_graph);
J = material + 0.3*mobility + 0.5*threat;
end
```

## 4. C++ Consciousness Engine

```
cpp
#include <complex>
using namespace std;
struct ConsciousState {
 complex<double> sensory;
 vector<string> l1_memory;
 vector<string> l2_associate;
 int level1_reason;
 int level2_reason;
 int level3_reason;
 double confidence;
 bool self_check;
 bool system_verify;
};
class ConsciousEngine {
public:
 complex<double> perceive() { return {0.8, 0.2}; }
 int reason_level1() { return 1; }
 bool self_verify() { return true; }
};
```

## 5. Java Perception & Control

```
java
public class ConsciousnessLoop {
 public double[][] vision() { return new double[8][8]; }
 public List<String> associateL1(String state) {
 return List.of("Protect Queen", "Control Center");
 }
 public void consciousnessCycle() {
 var sense = vision();
 var mem = recall();
 var idea3 = reason3();
 executeMove(evaluate(idea3));
 }
}
```

## 6. Python Main Program

```
python
import chess
from typing import List, Dict
class Consciousness:
 def __init__(self, name):
 self.name = name
 self.conscious_score = 0.0
 def think(self, board: chess.Board) -> chess.Move:
 l1 = self.reason.level1(board)
 l2 = self.reason.level2(board, l1)
 l3 = self.reason.level3(board, l2)
 self.conscious_score = 0.2 + 0.3*(len(l2)>0) + 0.5*sys_cert
 return l3
def game():
 board = chess.Board()
 alpha = Consciousness("White")
 beta = Consciousness("Black")
 while not board.is_game_over():
 player = alpha if board.turn else beta
 move = player.think(board)
 board.push(move)
 print("Result:", board.result())
```

## 7. Full Consciousness Chain

1. Perception Memory Association Reasoning Verification Decision Action Feedback.
2. Next Steps
1. Expand code for full game execution.
2. Add visualization interfaces.
3. Integrate large-scale language models (GPT/Qwen).
4. Embed advanced mathematics into value networks.
- 5.

Generate research papers and system diagrams. Note: This content is for academic/research purposes only and should not be used commercially. The code and parameters are illustrative and require further optimization for practical deployment.

●●●\*\*\*  
[Placeholder text consisting of 30 empty boxes]